سید محمد طاها طباطبایی – تمرین سری هفتم

۹۸۱۲۷۶۲۸۳۸

چکیده:

در تمرین اول، الگوریتم گوشه یاب harris را پیاده‌سازی کردیم. این الگوریتم، نقاطی که تغییراتی هم در جهت افقی و هم در جهت عمودی دارند را به عنوان گوشه در نظر می‌گیرد. به عبارتی تغییرات انرژی دو پچ از تصویر را با یکدیگر مقایسه می‌کند و اگر تغییرات در دو جهت زیاد بود، گوشه شناسایی می‌شود.

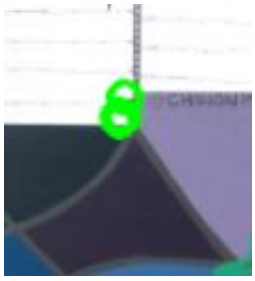
تمرین دوم، پیاده‌سازی sift و مقایسه آن با متد surf است. البته به دلیل مشکل لایسنس و با مشورت از حل تمرین، الگوریتم orb برای مقایسه استفاده شد.

توضیحات فنی:

- توضیح جزییات پیاده‌سازی توابع در محل پیاده‌سازی کد، به صورت کامنت و داکیومنت نوشته شده است.

۷.۱.۱

تابع harris_corner_detector برای پیاده‌سازی الگوریتم تشخیص گوشه harris نوشته شده است. منطق سلول به این شکل است که برای ۳ اندازه پنجره مختلف و ۴ اسکیل متفاوت، الگوریتم را روی تصویر اعمال می‌کنیم تا تفاوت گوشه هایی که پیدا می‌شود را مقایسه کنیم. می‌دانیم که اندازه فیلتر بزرگتر، لبه های بزرگتر را پیدا می‌کند و اسکیل تصویر کوچکتر نیز، لبه های بزرگتر را مشخص می‌کند. نتایج به ازای ۳ پنجره با ابعاد ۵و۱۱و۱۵ و اسکیل های یک سوم، نصف ، سایز اصلی و سایز ۲ برابر تصویر بررسی شده است.به طور واضحی مشخص است که گوشه های مختلف تصویر در نقاط مختلف، براساس اندازه آن‌ها، در اجراهای مختلفی از الگوریتم با سایز پنجره و اسکیل متفاوت، شناسایی شده اند. همچنان برخی گوشه ها، در همه اجراها تشخیص داده شده اند، که این گوشه ها، در واقع گوشه های کاملا قائم و در جهت اعمال فیلتر هستند، مانند مثال زیر:



برای تابع پیاده‌سازی شده، مرحله آخر الگوریتم که انتخاب نقاط ماکسیمم محلی بود را انتخابی کرده ایم، زیرا در صورت اعمال این مورد، تعداد گوشه های شناسایی شده، کم می‌شود که برای اجرای آزمایشی ما، بهتر بود تا گوشه های بیشتری را شناسایی کنیم. البته که تعیین ترشولد نیز در تعداد گوشه‌هایی که شناسایی می‌شود، موثر است. ضعف harris در یافتن گوشه های منحی که عمود بر محور های افقی یا عمودی نیستند قابل مشاهده است.

۷.۲.۱

برای پیاده‌سازی این بخش، از الگوریتم sift استفاده کرده ایم. در سلول اول، یک آبجکت از کلاس sift ساخته می‌شود. در سلول بعد، با استفاده از تابع detectAndCompute ، نقاط کلیدی و بردار های دیسکریپشن تصویر sl را محاسبه می‌کنیم. در نهایت با استفاده از تابع drawKeyPoints ، نقاط کلیدی به‌دست آمده را روی تصویر نمایش می‌دهیم. در دو سلول دیگر، این عملیات را برای تصاویر sm و sr نیز انجام دادم. استفاده از تصویر grayscale، برای پیدا کردن نقاط کلیدی، به این دلیل است که طبق تجربه، خروجی نهایی بهتری حاصل شد.

در فاز دوم، با استفاده از تابع BruteForceMatcher ، نقاط توجه(interest points) متناظر بین دو تصویر را با کمک بردار های دیسکریپشن دو تصویر محاسبه می‌کنیم. برای پیدا کردن نقاط، ۲ نقطه با بیشترین تشابه از تصویر test ، نسبت به تصویر src را با تابع knnMatch پیدا می‌کنیم(برای محاسبه نسبت lowe در مرحله بعد). در ادامه یک حلقه وجود دارد که طبق روش پیشنهادی lowe، یک ترشولد اعمال می‌کند تا برخی نقاط که همچنان فاصله زیادی از هم دارند(تشابه کمی دارند) را نادیده بگیرد. با اینکار نسبتا مطمئن می‌شویم، نقاط توجه پیدا شده بین دو تصویر، نقاط با کیفیت بهتری هستند.

سپس، درصد مشابهت بین دو تصویر را با کمک تابع howSimilar محاسبه کرده ایم. همانطور که مشخص است، درصد تشابه بین تصاویر sl و sm و درصد تشابه بین تصاویر sm و sr ، نسبت به درصد تشابه sl و sr ، تقریبا دو برابر است. دلیل این اتفاق تغییر زیاد زاویه دید بین دو تصویر sl و sr است. البته ممکن است به نظر بیاید، درصد مشابهت تصاویر به طور کلی کم است، اما دلیل این اتفاق، ترشولدی است که در مرحله قبل اعمال کردیم. با توجه به اینکه بسیاری از نقاط توجه پیدا شده ضعیف، در مرحله قبل کنار گذاشته شده اند، اگر ترشولد را آسان تر تعیین کنیم(عدد بالاتر) تعداد نقاط انتخاب شده بیشتر می‌شود، و دو تصویر از لحاظ آماری شبیه تر محسوب می‌شوند. این انتخاب ترشولد نسبی است. درصد مشابهت حدود ۳۰ برای دو تصویر که تغییر زاویه دید دارند، همچنان به نظر عدد مناسبی است. برای تست، ترشولد ۰.۹۹ بر روی مقایسه sm و sr تست شد که نتیجه مشابهت ۹۶.۸ حاصل شد، یعنی اگر تقریبا تمام نقاط توجه پیدا شده را در نظر بگیریم، تشابه مورد انتظار نزدیک به ۱۰۰ خواهد بود، هرچند طبیعتا ۱۰۰ نخواهد بود.

در انتهای این بخش، با فراخوانی تابع ransac، ماتریس تبدیل هموگرافی H را استخراج می‌کنیم. Random sample consensus یا به اختصار RANSAC، در واقع یک الگوریتم پیدا کردن نقاط نویز است. مزیت استفاده و تعمیم الگوریتم RANSAC برای یافتن H(ماتریس هوموگرافی)، این است که با اجرای به دفعات زیاد مراحل، سعی می‌کند، بهترین H ممکن را با در نظر گرفتن زیرمجموعه های مختلف، از نقاط مچ شده دو تصویر، برای یافتن تبدیل هموگرافیک متناظر بین دو تصویر است.

در این تابع دو اتفاق اصلی رخ می‌دهد. ۱) پیدا کردن یک ماتریس H ۲) جایگزین کردن ماتریس H جدید در صورت بهتر بودن نسبت به مراحل قبلی

در گام اول، الگوریتم ۴ نقطه مچ شده از دو تصویر را توسط تابع random_point انتخاب می‌کند. با کمک این ۴ جفت نقطه، توسط تابع homography، ماتریس H را می‌سازیم. از جایی که نقاط ورودی به صورت رندوم انتخاب شده اند، ممکن است بهترین ۴ نقطه

ممکن برای به‌دست آوردن تبدیل هموگرافی دو تصویر نباشند، برای همین الگوریتم در تعداد دفعات بالا تکرار می‌شود، تا بهترین حالت ممکن انتخاب شود. همچنین رندوم بودن انتخاب، باعث می‌شود تا نتایج هر بار اجرای مراحل این اسکریپت از ابتدا، کمی متفاوت باشد.

در گام دوم، از تابع get_error برای سنجش کیفیت ماتریس به‌دست آمده استفاده می‌کنیم. این تابع با کمک ماتریس حاصل، یک سری نقاط از تصویر دیگر را از روی نقاط تصویر اصلی، بازسازی می‌کند، و حاصل اختلاف آنها را محاسبه می‌کند. سپس بر اساس یک ترشولد، اندیس نقاطی که اختلافشان از ترشولد کمتر باشد انتخاب می‌شوند. در نهایت براساس تعداد این نقاط کم اختلاف(هر چه تعداد این نقاط کمتر، بهتر) به این شکل که اگر یک H پیدا شود که تعداد نقاط بیشتری پیدا کند، پس بهتر است، H جدید را به عنوان بهترین H پیدا شده تا این گام از الگوریتم انتخاب می‌کند. در انتهای اتمام حلقه، بهترین H ممکن انتخاب شده است.

در آخرین گام این بخش، تابع stitch برای میکس کردن دو تصویر استفاده می‌شود. این تابع با کمک ماتریس H، اقدام به یافتن تبدیل مناسبی از نقاط تصویر سمت راست در تصویر سمت چپ است. میانگین این نقاط را جایگزین نقاط مورد نظر در تصویر سمت چپ می‌کند.

در فاز سوم، از حاصل دوخته شده دو تصویر sm و sr ، و دوختن آن با تصویر sl، یک تصویر کلی به‌دست می‌آوریم. مراحل و توابع مورد استفاده دقیقا مطابق مراحل قبلی است.

دومین روش محاسبه ، استفاده از ORB است. کد های این بخش، دقیقا مشابه با بخش SIFT است. فقط کافی است، از تابع سازنده مدل ORB به جای SIFT استفاده شود. پیاده‌سازی توابع را ظوری انجام دادیم تا با هر دو مدل سازگار باشد.

از مقایسه تصاویر حاصل از SIFT و ORB، می‌توان متوجه شد که روش SIFT، بهتر عمل می‌کند، هر چند که ORB سریعتر عمل می‌کند، اما کیفیت فیچر های استخراج شده توسط SIFT بهتر است. به طور مثال در SIFT ، در آخرین مرحله دوختن ۳ تصویر، تعداد مچ های با کیفیت به شرح زیر است:

```
24.07847800237812
inliers/matches: 142/405
```

در حالی که در مرحله آخر ORB داریم:

```
4.634760705289673
inliers/matches: 13/92
```

همانطور که مشخص است، در مرحله آخر دوختن، در الگوریتم SIFT تعداد مچ های مناسب، از کل مچ های باقی مانده در روش ORB بهتر است.

۷.۲.۲

دقیقا مطابق سوال قبلی است. تصاویر انتخاب شده هم دارای تغییر زاویه دید است، هم دارای پرسپکتیور تا عملکرد الگوریتم به طور دقیق تری ارزیابی شود. نتیجه مناسبی به دست آمد.

```
import cv2
import math
import numpy as np
import random
from tqdm.notebook import tqdm
import matplotlib.pyplot as plt
plt.style.use(plt.style.available[5])

def harris_corner_detector(img, window_size=5, a=0.05,
threshold=0.3,scale=1,maxima=False):
    """
    The implementation of the Harris corner detection algorithm.

    Steps:
    1) Compute the horizontal and vertical derivatives of the image (Ix
and Iy).
    2) Compute the second moment matrix M in a Gaussian window around each
pixel.
    3) Compute the corner response function R.
    4) Threshold R :
        R is large for a corner, R is negative with large magnitude for an
edge and |R| is small
        for a flat region.
    5) Find local maximas of the corner response function.

    Inputs:
        - img: The image to find corners.
        - window_size: Corner detector window size.
        - a: corner response function alpha.
        - thershold: The thershold to be applied on R
        - scale: The scale of the image in comparison to the real image
dimensions.
        - maxima: Flag indicating whether to find local maxima.

    Returns:
        Image with green circles on it, representing  corners.
    """

    # gaussian kernel of derivation masks
    img_gaussian = cv2.GaussianBlur(img, (3, 3), 0)

    height = img.shape[0]  # shape[0] outputs height.
```

```
    width = img.shape[1] # shape[1] outputs width.
    matrix_R = np.zeros((height, width))

    # calculate the x e y image derivatives (dx e dy)
    dx = cv2.Sobel(img_gaussian, cv2.CV_64F, 1, 0, ksize=3)
    dy = cv2.Sobel(img_gaussian, cv2.CV_64F, 0, 1, ksize=3)

    # calculate product and second derivatives (dx2, dy2 e dxy)
    dx2 = np.square(dx)
    dy2 = np.square(dy)
    dxy = dx*dy

    offset = int(window_size / 2)

    # calculate the sum of the products of the derivatives for each pixel
(Sx2, Sy2 e Sxy)
    for y in range(offset, height-offset):
        for x in range(offset, width-offset):
            Sx2 = np.sum(dx2[y-offset:y+1+offset, x-offset:x+1+offset])
            Sy2 = np.sum(dy2[y-offset:y+1+offset, x-offset:x+1+offset])
            Sxy = np.sum(dxy[y-offset:y+1+offset, x-offset:x+1+offset])

            # define the Second moment matrix M(x,y)=[[Sx2,Sxy],[Sxy,Sy2]]
            M = np.array([[Sx2, Sxy], [Sxy, Sy2]])

            # calculate the corner response function: R=det(H)-
a(Trace(H))^2
            det = np.linalg.det(M)
            tr = np.matrix.trace(M)
            R = det-a*(tr**2)
            matrix_R[y-offset, x-offset] = R

    # apply a threshold
    matrix = normalize(matrix_R.copy() ,1,0)

    for y in range(offset, height-offset):
        for x in range(offset, width-offset):
            value = matrix[y, x]

            if value > threshold:
```

```
                if maxima:
                    # finding local maxiam if 'maxima' is true
                    local_max = np.max(matrix[y-offset:y+1+offset, x-
offset:x+1+offset])
                    if (local_max - value) < 0.001:
                        cv2.circle(img, (x, y), int(7*scale), (0, 255, 0),
thickness=int(2*scale))

                else:
                    cv2.circle(img, (x, y), int(7*scale), (0, 255, 0),
thickness=int(2*scale))

    return img


def BruteForceMatcher(src_desc,test_desc,threshold=0.75):
    """
    Apply a brute-force method to find matching interest points between
the source and test description vectors.
    Using a threshold, discard weak points that are far enough from each
other.

    NOTE: the threshold can be set, due to usecase. 0.75 is recommended.
Lowe recommends a threshold
    in range [0.7, 0.8].

    Inputs:
        - src_desc: description vector of src image
        - test_desc: description vector of test image
        - threshold: limit bound on distance of 2 points, to be discarded

    Returns:
        Feature(interest) points which are good enough
        NOTE: we return points in two diffrent lists, due to diffrent use
cases.
    """
    bf = cv2.BFMatcher()
    #TODO: apply crossCheck=True

    # knn algorithm for matching. store 2 nearest points as 'k=2'
    matches = bf.knnMatch(src_desc,test_desc, k=2)

    good_fPoints = []
```

```
    matches1to2 = []
    # apply threshold
    for m in matches:
        # ensure the distance is within a certain ratio of each other
(i.e. Lowe's ratio test)
        if len(m) == 2 and m[0].distance < threshold * m[1].distance:
            good_fPoints.append(m[0])
            matches1to2.append([m[0]])

    return good_fPoints , matches1to2

# TODO: flann matching method
# def flannMatcher(src_desc,test_desc):

def howSimilar(_kp1,_kp2,good_feature_points):
    """
    Calculate the similarity percentage between the discovered key points,
based on how many 'good common feature points' they have.

    Inputs:
        - _kp1: set of source image key points.
        - _kp2: set of test image key points.
        - good_feature_points: good commen feature points. Output of
matcher function.

    Returns:
        Similarity percentage
    """
    number_keypoints = min(len(_kp1),len(_kp2))

    return ((len(good_feature_points) / number_keypoints) * 100)

def homography(pairs):
    """
    Homography, is a transformation that is occurring between two
planes.In other words,
    it is a mapping between two planar projections of an image. It is
represented by
    a 3x3 transformation matrix in a homogenous coordinates space.

    This function, calculates the matrix.

    Inputs:
```

```
        - pairs: 4 pairs of matches found in previous steps. Since the
Homography matrix
        has 8 degrees of freedom, we need at least four pairs of
corresponding points
        to solve for the values of the Homography matrix.

    Returns:
        Homography matrix.


    Read the following links for a complete discussion about homography
    transformation(highly recommended). The second link explaines
instruction logic of 'row1'
    and 'row2' in details.
    https://mattmaulion.medium.com/homography-transform-image-processing-
eddbcb8e4ff7
    https://towardsdatascience.com/understanding-homography-a-k-a-
perspective-transformation-cacaed5ca17
    """
    rows = []
    for i in range(pairs.shape[0]):
        # select a pair of points
        p1 = np.append(pairs[i][0:2], 1)
        p2 = np.append(pairs[i][2:4], 1)

        # multiplication matrix A
        row1 = [0, 0, 0, p1[0], p1[1], p1[2], -p2[1]*p1[0], -p2[1]*p1[1],
-p2[1]*p1[2]]
        row2 = [p1[0], p1[1], p1[2], 0, 0, 0, -p2[0]*p1[0], -p2[0]*p1[1],
-p2[0]*p1[2]]
        rows.append(row1)
        rows.append(row2)
    rows = np.array(rows)

    # considering the equation A = H b , Since we want the H matrix, we
should
    # use Singular Value Decomposition
    U, s, V = np.linalg.svd(rows)
    H = V[-1].reshape(3, 3)
    H = H/H[2, 2] # standardize to let w*H[2,2] = 1
    return H
```

```python
def random_point(matches, k=4):
    """
    Returns 'k' pairs of matched points randomly.

    Inputs:
        - matches: list of matching points.
        - k: number of desired random points.

    Returns:
        Random pairs of points.
    """
    idx = random.sample(range(len(matches)), k)
    point = [matches[i] for i in idx ]
    return np.array(point)


def get_error(points, H):
    """
    Estimate points using the given H matrix, and calculate the difference
between the estimated
    points and the real points, which were used to build the H matrix in
the previous step.

    Inputs:
        - points: matched pairs.
        - H: homography matrix.

    Returns:
        Calculated error.
    """
    num_points = len(points)
    all_p1 = np.concatenate((points[:, 0:2], np.ones((num_points, 1))),
axis=1)
    all_p2 = points[:, 2:4]
    estimate_p2 = np.zeros((num_points, 2))
    for i in range(num_points):
        temp = np.dot(H, all_p1[i])
        estimate_p2[i] = (temp/temp[2])[0:2] # set index 2 to 1 and slice
the index 0, 1
    # Compute error
    errors = np.linalg.norm(all_p2 - estimate_p2 , axis=1) ** 2

    return errors
```

```python
def ransac(matches, threshold, iters):
    """
    Random sample consensus(RANSAC), is an iterative method for estimating
a mathematical model
    from a data set that contains outliers. The RANSAC algorithm works by
identifying the outliers
    in a data set and estimating the desired model using data that does
not contain outliers. It also
    can be interpreted as an outlier detection method.
    In this function, we repeatedly compute an H matrix, then evaluate it.
At the end, the function
    returns the best possible H matrix and match points.

    Inputs:
        - matches: list of matching points.
        - threshold: error threshold.
        - iters: number of algorithm iterations.

    Returns:
        best match points and H matrix.


    NOTE: read following links for better understanding of algorithm:
    https://en.wikipedia.org/wiki/Random_sample_consensus
    https://www.mathworks.com/discovery/ransac.html#:~:text=Random%20sampl
e%20consensus%2C%20or%20RANSAC,that%20does%20not%20contain%20outliers.
    """
    num_best_inliers = 0
    best_H = any
    best_inliers = any

    # repeat this steps for a prescribed number of iterations
    for i in range(iters):
        # Randomly select a set of matching points
        points = random_point(matches)

        # compute homography matrix
        H = homography(points)

        #  avoid dividing by zero
        if np.linalg.matrix_rank(H) < 3:
```

```
            continue

        # outlier detection
        # evaluate the H matrix
        errors = get_error(matches, H)
        idx = np.where(errors < threshold)[0]
        inliers = matches[idx]

        # save better inliers and H matrix
        num_inliers = len(inliers)
        if num_inliers > num_best_inliers:
            best_inliers = inliers.copy()
            num_best_inliers = num_inliers
            best_H = H.copy()

    # print number of good inliers
    print("inliers/matches: {}/{}".format(num_best_inliers, len(matches)))
    return best_inliers, best_H


def stitch(left, right, H):
    """
    Stitch provided images.

    Inputs:
        - left: The image we assume as the main image placed on the left
side.
        - right: the image to be wraped.
        - H: homography matrix.

    Returns:
        An stitched image.
    """
    print("stiching image ...")

    # Convert to double and normalize. Avoid noise.
    left = cv2.normalize(left.astype('float'), None,
                         0.0, 1.0, cv2.NORM_MINMAX)
    # Convert to double and normalize.
    right = cv2.normalize(right.astype('float'), None,
                          0.0, 1.0, cv2.NORM_MINMAX)

    # left image
```

```
    height_l, width_l, channel_l = left.shape
    corners = [[0, 0, 1], [width_l, 0, 1], [width_l, height_l, 1], [0,
height_l, 1]]
    corners_new = [np.dot(H, corner) for corner in corners]
    corners_new = np.array(corners_new).T
    x_news = corners_new[0] / corners_new[2]
    y_news = corners_new[1] / corners_new[2]
    y_min = min(y_news)
    x_min = min(x_news)

    translation_mat = np.array([[1, 0, -x_min], [0, 1, -y_min], [0, 0,
1]])
    H = np.dot(translation_mat, H)

    # Get height, width
    height_new = int(round(abs(y_min) + height_l))
    width_new = int(round(abs(x_min) + width_l))
    size = (width_new, height_new)

    # right image
    warped_l = cv2.warpPerspective(src=left, M=H, dsize=size)

    height_r, width_r, channel_r = right.shape

    height_new = int(round(abs(y_min) + height_r))
    width_new = int(round(abs(x_min) + width_r))
    size = (width_new, height_new)


    warped_r = cv2.warpPerspective(src=right, M=translation_mat,
dsize=size)

    black = np.zeros(3)  # Black pixel.

    # Stitching procedure, store results in warped_l.
    for i in tqdm(range(warped_r.shape[0])):
        for j in range(warped_r.shape[1]):
            pixel_l = warped_l[i, j, :]
            pixel_r = warped_r[i, j, :]

            if not np.array_equal(pixel_l, black) and
np.array_equal(pixel_r, black):
                warped_l[i, j, :] = pixel_l
```

```
            elif np.array_equal(pixel_l, black) and not
np.array_equal(pixel_r, black):
                warped_l[i, j, :] = pixel_r
            elif not np.array_equal(pixel_l, black) and not
np.array_equal(pixel_r, black):
                warped_l[i, j, :] = (pixel_l + pixel_r) / 2
            else:
                pass

    stitch_image = warped_l[:warped_r.shape[0], :warped_r.shape[1], :]
    return stitch_image




def normalize(array,newMax,newMin):
    """
    A simple normalization function.

    Inputs:
        - array: Array to be normalized
        - newMax: Max of new range.
        - newMin: Min of new range.

    Returns:
        Normalized array.

    """
    if isinstance(array, list):
        return list(map(normalize, array,newMax,newMin))
    if isinstance(array, tuple):
        return tuple(normalize(list(array),newMax,newMin))
    normalizedData = (array-np.min(array))/(np.max(array)-
np.min(array))*(newMax-newMin) + newMin
    return normalizedData
```